Фасад

Слайд 2

Фасад — це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

Слайд 3

Проблема

Вашому коду доводиться працювати з великою кількістю об’єктів певної складної бібліотеки чи фреймворка. Ви повинні самостійно ініціалізувати ці об’єкти, стежити за правильним порядком залежностей тощо.

В результаті бізнес-логіка ваших класів тісно переплітається з деталями реалізації сторонніх класів. Такий код досить складно розуміти та підтримувати.

Слайд 4

Рішення

Фасад — це простий інтерфейс для роботи зі складною підсистемою, яка містить безліч класів. Фасад може бути спрощеним відображенням системи, що не має 100% тієї функціональності, якої можна було б досягти, використовуючи складну підсистему безпосередньо. Разом з тим, він надає саме ті «фічі», які потрібні клієнтові, і приховує все інше.

Фасад корисний у тому випадку, якщо ви використовуєте якусь складну бібліотеку з безліччю рухомих частин, з яких вам потрібна тільки частина.

Наприклад, програма, що заливає в соціальні мережі відео з кошенятками, може використовувати професійну бібліотеку для стискання відео, але все, що потрібно клієнтському коду цієї програми, — це простий метод `encode(filename, format)`. Створивши клас з таким методом, ви реалізуєте свій перший фасад.

Слайд 5

Аналогія з життя: приклад замовлення через телефон

Коли ви телефонуєте до магазину і робите замовлення, співробітник служби підтримки є вашим фасадом до всіх служб і відділів магазину. Він надає вам спрощений інтерфейс до системи створення замовлення, платіжної системи та відділу доставки.

Слайд 6

Структура

1. Фасад надає швидкий доступ до певної функціональності підсистеми. Він «знає», яким класам потрібно переадресувати запит, і які дані для цього потрібні.

2. Додатковий фасад можна ввести, щоб не захаращувати єдиний фасад різнорідною функціональністю. Він може використовуватися як клієнтом, так й іншими фасадами.

3. Складна підсистема має безліч різноманітних класів. Для того, щоб примусити усіх їх щось робити, потрібно знати подробиці влаштування підсистеми, порядок ініціалізації об’єктів та інші деталі. Класи підсистеми не знають про існування фасаду і працюють один з одним безпосередньо.

4. Клієнт використовує фасад замість безпосередньої роботи з об’єктами складної підсистеми.

Слайд 7

Псевдокод: приклад ізоляції множини залежностей в одному фасаді

У цьому прикладі Фасад спрощує роботу зі складним фреймворком конвертації відео.
Замість безпосередньої роботи з дюжиною класів, фасад надає коду програми єдиний метод для конвертації відео, який сам подбає про те, щоб правильно налаштувати потрібні об’єкти фреймворку і отримати необхідний результат.

Слайд 8

Застосування

Якщо вам потрібно надати простий або урізаний інтерфейс до складної підсистеми.
Часто підсистеми ускладнюються в міру розвитку програми. Застосування більшості патернів призводить до появи менших класів, але у великій кількості. Таку підсистему простіше використовувати повторно, налаштовуючи її кожен раз під конкретні потреби, але, разом з тим, використовувати таку підсистему без налаштовування важче. Фасад пропонує певний вид системи за замовчуванням, який влаштовує більшість клієнтів.

Слайд 9

Якщо ви хочете розкласти підсистему на окремі рівні.
Використовуйте фасади для визначення точок входу на кожен рівень підсистеми. Якщо підсистеми залежать одна від одної, тоді залежність можна спростити, дозволивши підсистемам обмінюватися інформацією тільки через фасади.

Наприклад, візьмемо ту ж саму складну систему конвертації відео. Ви хочете розбити її на окремі шари для роботи з аудіо й відео. Можна спробувати створити фасад для кожної з цих частин і примусити класи аудіо та відео обробки спілкуватися один з одним через ці фасади, а не безпосередньо.

Слайд 10

Кроки реалізації

1. Визначимо, чи можна створити більш простий інтерфейс, ніж той, який надає складна підсистема. Ми на правильному шляху, якщо цей інтерфейс позбавить клієнта від необхідності знати подробиці підсистеми.
2. Створімо клас фасаду, що реалізує цей інтерфейс. Він повинен переадресовувати виклики клієнта потрібним об’єктам підсистеми. Фасад повинен буде подбати про те, щоб правильно ініціалізувати об’єкти підсистеми.
3. Ми отримаємо максимум користі, якщо клієнт працюватиме тільки з фасадом. В такому випадку зміни в підсистемі стосуватимуться тільки коду фасаду, а клієнтський код залишиться робочим.

4. Якщо відповідальність фасаду стає розмитою, можемо подумати про введення додаткових фасадів.

Слайд 11

Переваги та недоліки

Переваги:
Ізолює клієнтів від компонентів складної підсистеми.

Недоліки:
Фасад ризикує стати божественим об’єктом, прив’язаним до всіх класів програми.

Слайд 12

Відносини з іншими патернами

- **Фасад** задає новий інтерфейс, тоді як **Адаптер** повторно використовує старий. Адаптер обгортає тільки один клас, а Фасад обгортає цілу підсистему. Крім того, Адаптер дозволяє двом існуючим інтерфейсам працювати спільно, замість того, щоб визначити повністю новий.

- **Абстрактна фабрика** може бути використана замість **Фасаду** для того, щоб приховати платформо-залежні класи.

- **Легковаговик** показує, як створювати багато дрібних об’єктів, а **Фасад** показує, як створити один об’єкт, який відображає цілу підсистему.

- **Посередник** та **Фасад** схожі тим, що намагаються організувати роботу багатьох існуючих класів.

    - ***Фасад*** створює спрощений інтерфейс підсистеми, не вносячи в неї жодної додаткової функціональності. Сама підсистема не знає про існування Фасаду. Класи підсистеми спілкуються один з одним безпосередньо.
    - ***Посередник*** централізує спілкування між компонентами системи. Компоненти системи знають тільки про існування Посередника, у них немає прямого доступу до інших компонентів.

Слайд 13

- **Фасад** можна зробити **Одинаком**, оскільки зазвичай потрібен тільки один об’єкт-фасад.

- **Фасад** схожий на **Замісник** тим, що замінює складну підсистему та може сам її ініціалізувати. Але, на відміну від Фасаду, Замісник має такий самий інтерфейс, що і його службовий об’єкт, завдяки чому їх можна взаємозаміняти.

Слайд 14

Концептуальний приклад

Мій приклад показує структуру патерна **Фасад**, а саме — з яких класів він складається, які ролі ці класи виконують і як вони взаємодіють один з одним.

Код демонструє патерн проектування "Фасад" (Facade), який спрощує взаємодію зі складними системами або підсистемами, надаючи більш простий інтерфейс.

Основні компоненти коду:

1. **Клас Facade**:

- Ініціалізація: Конструктор приймає два параметри (`subsystem1` і `subsystem2`), які можуть бути передані ззовні або створені всередині класу, якщо не передані.

- Метод `operation`: Цей метод об'єднує операції з різних підсистем і повертає їх результати у вигляді рядка. Він спочатку ініціалізує підсистеми, потім викликає методи підсистем для виконання дій.

2. **Підсистеми**:

Subsystem1 і Subsystem2 мають свої методи (`operation1`, `operation_n`, `operation_z`), які виконують специфічні дії. Ці класи представляють складні частини системи, з якими клієнт може взаємодіяти.

3. **Клієнтський код**:

- `client_code(facade)`: Ця функція приймає об'єкт фасаду і викликає його метод `operation`, щоб отримати результат, який клієнт може легко обробити. Це дозволяє клієнту взаємодіяти з підсистемами через простий інтерфейс фасаду, не знаючи про внутрішню складність.

Переваги використання фасаду:

- Спрощення взаємодії: Клієнт працює з фасадом, а не зі складними підсистемами, що робить код більш читабельним і легким в обслуговуванні.
- Схованість складності: Фасад управляє життєвим циклом підсистем і їх взаємодіями, що захищає клієнта від необхідності знати деталі реалізації.
- Гнучкість: Якщо в майбутньому потрібно буде змінити або розширити функціональність, це можна зробити, не торкаючись клієнтського коду.

Таким чином, патерн "Фасад" допомагає управляти складністю програмних систем, забезпечуючи зручний інтерфейс для взаємодії з ними.

Слайд 15

Дякую за увагу